FC

ARPA ORDER NO. 2223

ISI/RR-76-48
August 1976

12

John V. Guttag
USC Computer Science Department

Ellis Horowitz
USC Computer Science Department

David R. Musser
USC Information Sciences Institute

Abstract Data Types and Software Validation

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. ~~REPORT NUMBER~~ 14 ISI/RR-76-48	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. ~~TITLE (and Subtitle)~~ 6 ABSTRACT DATA TYPES AND SOFTWARE VALIDATION,		5. ~~TYPE OF REPORT & PERIOD COVERED~~ 9 Research Report
		6. ~~PERFORMING ORG. REPORT NUMBER~~
7. ~~AUTHOR(s)~~ 10 John V. Guttag, USC Ellis Horowitz, USC David R. Musser, ISI		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. ~~REPORT DATE~~ 11 Aug[illegible] [illegible]76
		13. ~~NUMBER OF PAGES~~ 12 50p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

15 DAHC15-72-C-Ø308,

18. SUPPLEMENTARY NOTES

ARPA Order-2223

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract data type, correctness proof, data type, data structure, specification, software specification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407952

VB

20. ABSTRACT

A data abstraction can be naturally expressed using algebraic axioms, whose virtue is that they permit a representation-independent formal specification of a data type. A moderately complex example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is two-fold. First, it is shown how the use of algebraic axiomatizations can significantly improve the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of a large software system at design time, long before a conventional implementation is accomplished. Extensions to the formalism which permit the handling of errors and procedures with side effects are also given.

ARPA ORDER NO. 2223

ISI/RR-76-48
August 1976

John V. Guttag
USC Computer Science Department

Ellis Horowitz
USC Computer Science Department

David R. Musser
USC Information Sciences Institute

Abstract Data Types and Software Validation

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research was supported in part by the National Science Foundation under Grant No. MCS76-06089 and by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223, Program Code Nos. 3D30 and 3P10.

PRECEDING PAGE BLANK-NOT FILMED

ABSTRACT

A data abstraction can be naturally expressed using algebraic axioms, whose virtue is that they permit a representation-independent formal specification of a data type. A moderately complex example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can significantly improve the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of a large software system at design time, long before a conventional implementation is accomplished. Extensions to the formalism which permit the handling of errors and procedures with side effects are also given.

1. *INTRODUCTION*

The key problem in the design and validation of large software systems is reducing the amount of complexity or detail that must be considered at any one time; two common and effective solution methods are decomposition and abstraction. One decomposes a task by factoring it into two or more separable subtasks. Unfortunately, for many problems the separable subtasks are still too complex to be mastered *in toto*. The complexity of this sort of problem can be reduced via abstraction. By providing a mechanism for separating those attributes of an object or event that are relevant in a given context from those that are not, abstraction serves to reduce the amount of detail that must be comprehended at any one time.

If one is to make full use of abstraction, it is critical to have available a good notation for expressing abstractions. It is obvious that a reasonable language is a prerequisite to communicating something as intangible as an abstraction; it is less obvious, but equally true, that a reasonable language is a prerequisite to the creation of such abstractions. Even if a language need not be available for the initial formulation of an abstraction (an argument we leave to psychologists and linguists), it is certainly necessary if the abstraction is to be retained and developed over any significant period of time.

An informal language, such as a natural language, is not always efficient--either for the communication of abstractions or for their creation. Unlike formal languages, informal languages do not force precision. In fact, only with great care and expertise is it possible to write precise specifications in a notation as informal as a natural language. The problem is compounded when one attempts to use an informal notation as a mechanism for communicating an abstraction to others. Not only might the specification be underdefined--hence ambiguous--but the very language in which the specification is stated will almost certainly be ambiguous as well. If all goes well, someone will perceive the ambiguity, and it will be resolved. More often, each of those involved will merely form his own different conception of the abstraction, which creates interface problems.

Of course, the use of a formal language is no guarantee that specifications will be unambiguous or even consistent. It is, for example, quite possible to specify ambiguous grammars or empty languages in BNF. What a formal language does provide are objective criteria for recognizing ambiguity and inconsistency, thus increasing the likelihood that such failings in a specification will be recognized.

A formal specification also provides a necessary prerequisite to formal proofs of program correctness. Such proofs are, in fact, proofs of program consistency. That is to say, we prove that a program is consistent with its specification. If this proof is to be formal, than we must have available a formal specification of what the program is supposed to do.

A recent trend in programming is the development of abstract data types or data abstraction. In data abstraction, a number of functional abstractions are grouped together. The clustered operations are related by the fact that they, and only they, operate on a particular class or type of object. Some typical data abstractions are a symbol table, a priority queue, and a set.

In this report we shall present a notation for describing data abstractions, which we call algebraic axioms. Our experience indicates that the resulting specification of a data abstraction using algebraic axioms is both rigorous and easy to comprehend; see [Guttag76b]. In order to show how these specifications can be used during the design process, we exhibit, in Sections 2 and 3, their use in the creation of a symbol table which allows for block structure. This example is sufficiently rich that it reveals most of the facets of using the notation when coupled with the top-down design process.

However, the point we wish to stress in this report is not design but the use of algebraic axioms for proofs of correctness and for program testing. In Section 4 we show how this axiomatic technique can be profitably employed to prove the correctness of an implementation of a data abstraction. The strength of the technique is that it factors the proving process into distinct, manageable stages; further, it simplifies the proof at each stage. In Section 5 we discuss an automated system which processes an algebraic axiomatization of a data abstraction in such a way that correctness proofs can be carried out semiautomatically; in addition, programs may be tested before an implementation into a conventional programming language is achieved. This coupling of testing and correctness is a valuable byproduct of the algebraic axiom approach and is a strong argument for its worth. Section 6 addresses some proposed extensions to algebraic axiomatizations designed to handle procedures with side effects and the specification of errors.

2. *DEFINITIONS, CONCEPTS, AND EXAMPLES*

Rather than presenting formal definitions of abstract data types and related concepts, we prefer to give informal and (hopefully) intuitively appealing definitions and to illustrate the main ideas with a number of examples. We shall view a *data type* T as a class of values and a collection of operations on the values. If the properties of the operations are specified only by axioms, we call T an *abstract data type* or a *data abstraction*. An *implementation* of a data abstraction is an assignment of meaning to the values and operations in terms of the values and operations of another data type or set of data types. A *correct implementation* of a data abstraction is one which satisfies the axioms.

An *algebraic axiom specification* of a data type T consists of a *syntactic* and a *semantic* description; the syntactic specification defines the names, domains and ranges of the operations of T, and the semantic specification contains a set of axioms in the form of equations which relate the operations of T to each other. The term "algebraic" is appropriate because the values and axioms can be regarded as an abstract algebra. [Goguen75] and [Zilles75] have strongly emphasized the algebraic approach, developing the theory of abstract data types as an application of heterogeneous algebras. With this approach implementations are treated like other algebras, and the problem of showing an implementation to be correct is treated as one of showing the existence of a homomorphic mapping from one algebra to another. We shall in this report de-emphasize the explicit use of algebraic terminology, preferring instead the terminology of programming. In spite of this difference in terminology, there are many similarities between our approach and the more purely algebraic approach. A significant technical difference which does exist between the two approaches will be discussed in Section 4.5.

The choice of a language in which to express the specifications is important. We must be able to express the relationships among the operations both precisely and clearly.

In addition, the specification language itself must be axiomatically defined to facilitate correctness proofs. We begin by assuming a base language with five primitives: functional composition, an equality relation (=), two distinct constants (TRUE and FALSE), and an unbounded supply of free variables. From these primitives one can construct an arbitrarily complex specification language, for once an operation has been defined in terms of the primitives it may be added to the specification language. An IF-THEN-ELSE operation, for example, may be defined by the axioms

```
IF-THEN-ELSE(TRUE,q,r) = q,
IF-THEN-ELSE(FALSE,q,r) = r.
```

Throughout this report we shall assume that the expression IF-THEN-ELSE(b,q,r), which we shall write as IF b THEN q ELSE r, is part of the specification language. We shall also assume the availability of infix Boolean operators such as $\wedge$, $\vee$, $\neg$, $\supset$, $\equiv$, etc. The axiomatization of these operators in terms of the IF-THEN-ELSE operator will be given in Section 4. Finally, we allow for the conventional operations on integers: PLUS, MINUS, TIMES, DIV, and MOD (conventional infix operators are used when convenient).

2.1 *STACK EXAMPLE*

One of the simplest examples of an abstract data type is the unbounded Stack type.

```
type  Stack[elementtype: Type]

syntax

            NEWSTACK → Stack,
            PUSH(Stack,elementtype) → Stack,
            POP(Stack) → Stack,
            TOP(Stack) → elementtype ∪ {UNDEFINED},
            ISNEW(Stack) → Boolean,
            REPLACE(Stack,elementtype) → Stack.

semantics

      declare stk:Stack, elm:elementtype;
            POP(NEWSTACK) = NEWSTACK,
            POP(PUSH(stk,elm)) = stk,
            TOP(NEWSTACK) = UNDEFINED,
            TOP(PUSH(stk,elm)) = elm,
            ISNEW(NEWSTACK) = TRUE,
            ISNEW(PUSH(stk,elm)) = FALSE,
            REPLACE(stk,elm) = PUSH(POP(stk),elm).
```

Figure 2.1: Stack Data Type

In the example of Figure 2.1 we have defined a data type Stack with six operations via a syntactic specification of these operations and a semantic specification which is a set of seven equations relating the operations. Certain notational conventions of this example will be used throughout. Operation names are written using all capital letters. The name of a data type begins with a capital. In the equations the lower case symbols are free variables ranging over the domains indicated, e.g., stk ranges over the Stack type. The symbol elementtype is a variable ranging over the set of types and the variable elm ranges over elementtype. This says that we can have a Stack of any type of elements (but all must be of the same type); what we have defined is thus not a single type, but rather a *type schema*. The binding of elementtype to a particular type, e.g. Stack[Integer], reduces the schema to a specification of a single type. With the syntactic specification of the operations, each of the expressions in the axiomatic equations can be checked for well-formedness in the sense that each operator is applied to the correct number of arguments and each argument has the correct type.

The equations are statements of fact (axioms) relating the values which are created by the operations, e.g., the equation

$$\text{TOP(PUSH(stk,elm))} = \text{elm}$$

means that for any Stack value stk and any elementtype value elm, the result of PUSH(stk,elm) is a Stack value, stk1, such that TOP(stk1) yields the value elm.

In viewing the equations in this way, we are not required to give any particular interpretation to the values; their "useful" properties can be derived solely from the relations determined by the axioms. Thus, in designing computer implementations of the operations, we are free to represent the values in many different ways.

A typical implementation is shown in Figure 2.2. Each Stack value is represented by a structure with two components--an unbounded array, whose components are of type elementtype, and an integer indicating the position in the array of the top element of the stack. We intend that all of the operations be purely "functional" or "applicative," i.e., have no side effects, in accordance with the requirements of the basic theoretical framework for algebraic specifications. This can imply an unrealistic degree of inefficiency for implementations. In the Stack implementation of Figure 2.2, for example, the implementation of PUSH must involve copying the array component as well as the Integer component of the Stack representation. The basic framework can be extended to permit specification of operations with side effects, so that the obvious efficient implementations are possible. However, since the exposition of our proof techniques is facilitated by this restriction, we will continue to assume no side effects for now; we shall discuss this extension in Section 6.

Figure 2.2 shows a Stack implementation written without assignment to variables. For the array assignment we have used the assignment operator as defined by McCarthy. For the type schema Array[domain:Type, range:Type], ASSIGN(arr,t,elm) means the array identical to arr except possibly in the t-th position, where the value is elm [McCarthy63]. ACCESS(arr,t) returns the value in position t of the array arr. (For a complete specification of type array see Figure 3.3.)

```
representation STAK(Array[Integer,elementtype],Integer) → Stack[elementtype],

programs

   declare arr:Array, t:Integer, elm:elementtype;
          NEWSTACK = STAK(NEWARRAY,0),
          PUSH(STAK(arr,t),elm) = STAK(ASSIGN(arr,t+1,elm),t+1),
          POP(STAK(arr,t)) = IF t=0 THEN STAK(arr,0) ELSE STAK(arr,t-1),
          TOP(STAK(arr,t)) = ACCESS(arr,t),
          ISNEW(STAK(arr,t)) = (t=0),
          REPLACE(STAK(arr,t),elm) =
                   IF t=0 THEN STAK(ASSIGN(arr,1,elm),1)
                            ELSE STAK(ASSIGN(arr,t,elm),t).
```

Figure 2.2:* *An Implementation of the Stack Data Type with (Array,Integer) Pairs

We have divided the implementation into a *representation* part and a *programs* part. This corresponds to the division of the specification into a syntactic part and a semantic part. STAK is an operation whose domain is the cross product of types Array and Integer and whose range is type Stack.

In this report the language used to express programs is the same as for describing axioms. Though we recognize that a richer language is usually more desirable, we have chosen to restrict ourselves here for several reasons. Most importantly, the proof procedure described in Section 4 derives much of its simplicity from the use of this restricted set of constructs. Since all conventional programming language features can be automatically translated into our basic set (see [Manna74]), no real advantage is lost. We are able to avoid issues of language design, concentrating on how the basic command set can be axiomatized and used for correctness proofs and to synthesize implementations. The clarity of the basic language for axiomatizing data types has been shown in [Guttag76b] and [Horowitz76].

The correctness of an implementation of a data type can be proved by showing that each axiom of the semantic specification is satisfied by the programs. As a particularly simple example of such a proof, consider the fourth Stack axiom. Assuming stk=STAK(arr,t),

```
TOP(PUSH(stk,elm)) = TOP(PUSH(STAK(arr,t),elm))
                   = TOP(STAK(ASSIGN(arr,t+1,elm),t+1))
                   = ACCESS(ASSIGN(arr,t+1,elm),t+1)
                   = elm.
```

The other Stack axioms can be shown to be satisfied in a similar manner, although not quite so straightforwardly. The complications that arise will be dealt with in Section 4, which discusses verification of implementations in detail.

2.2 *PROGRAMS AS AXIOMS AND AXIOMS AS PROGRAMS*

In the discussion of the implementation for the Stack data type, we described STAK(arr,t) as a pair whose first component is an Array and second component is an Integer, and viewed equations such as

TOP(STAK(arr,t)) = ACCESS(arr,t)

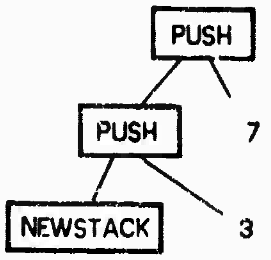
as definitions of programs for operating on the STAK pairs. Suppose, however, that we now view STAK as an *operation* whose syntactic specification is STAK(Array[Integer,elementtype],Integer)→Stack[elementtype]. Then the above equation for TOP and the other program equations can be viewed as *axioms* that comprise a semantic specification for STAK. Viewing it as an axiom, we would read the TOP equation as "if stk is the result of applying STAK to an Array arr and an Integer t, the value returned by TOP(stk) is ACCESS(arr,t)."

As an axiomatic specification of the Stack data type, the implementation of Figure 2.2 is inferior to the specification of Figure 2.1 in that it is not self-contained (it requires knowledge of properties of Arrays and Integers). We have called attention to the view of programs as axioms mainly because it suggests a *duality* between programs and axioms whose other half--*axioms as programs*--can be very fruitfully exploited.

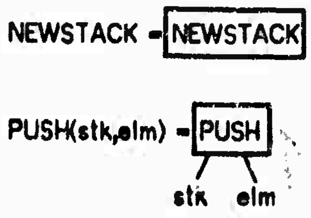
We can, in fact, view the axioms of Figure 2.1 as programs simply by regarding NEWSTACK and PUSH(stk,elm) as *trees* rather than operations. All structures built with NEWSTACK and PUSH can be pictured as trees. For example,

PUSH(PUSH(NEWSTACK,3),7)

can be diagrammed as



The Stack axioms can be viewed as defining operations which produce and access such tree structures:



$$POP(\boxed{NEWSTACK}) = \boxed{NEWSTACK}$$

$$POP(\underset{stk \quad elm}{\boxed{PUSH}}) = stk$$

etc.

The two equations for POP together define POP as an operation that first checks which kind of node it is given and then proceeds accordingly. This is an example of a *direct implementation.*

Direct implementations are useful in a number of ways. First, the concept of a direct implementation can serve as an aid to constructing specifications; i.e., one can try to write the semantic axioms so that they can serve as programs operating on tree structures. If this can be done, and one has a compiler for actually producing operational implementations of such programs, then one can experiment with the operations, testing to a limited extent whether they have the properties intended. More importantly, one can also test high-level algorithms programmed in terms of the data type before fixing upon a particular implementation of the data type. Thus, a true top-down design methodology can be achieved.

Further examples of direct implementations will appear in Sections 2.3 and 3. Many examples of algebraic axiomatizations of data types with explanations of their direct implementations appear in [Guttag76b]. Section 5 will contain a more detailed discussion of direct implementations and the related idea of "reduction systems."

2.3 *A RICHER EXAMPLE*

The Stack data type is too simple, in a number of respects, to properly illustrate the properties and uses of algebraic axiom specifications. The equations have too simple a form and the usable implementations are too straightforward to have "interesting" proofs of correctness. A much richer example is provided by the symbol table data type, which was first specified algebraically in [Guttag75], [Guttag76a]. In this example we deal with a common but nontrivial data structuring problem, the design of a symbol table for a compiler for a block-structured language. We wish to specify and implement a set of operations for maintaining the symbol table during compilation of a program. An informal specification of the operations might be as follows:

INIT:	Allocate and initialize the symbol table.
ENTERBLOCK:	Prepare a new local naming scope.
ADDID:	Add an identifier and its attributes to the symbol table.
LEAVEBLOCK:	Discard entries from the most current scope and re-establish the next outer scope.

ISINBLOCK:	Has a specified identifier already been declared in this scope (used to check for duplicate declaration)?
RETRIEVE:	Return the attributes associated with the most local definition of a specified identifier.

The formal specification which we shall adopt is given in Figure 2.3.

```
type Symboltable

syntax

        INIT → Symboltable,
        ENTERBLOCK(Symboltable) → Symboltable,
        ADDID(Symboltable,Identifier,Attributelist) → Symboltable,
        LEAVEBLOCK(Symboltable) → Symboltable,
        ISINBLOCK(Symboltable,Identifier) → Boolean,
        RETRIEVE(Symboltable,Identifier) → Attributelist ∪ {UNDEFINED}.

semantics

  declare symtab:Symboltable, id,id1: Identifier, attrlist: Attributelist;
      1)  LEAVEBLOCK(INIT) = INIT,
      2)  LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab,
      3)  LEAVEBLOCK(ADDID(symtab,id,attrlist)) = LEAVEBLOCK(symtab),
      4)  ISINBLOCK(INIT,id) = FALSE,
      5)  ISINBLOCK(ENTERBLOCK(symtab),id) = FALSE,
      6)  ISINBLOCK(ADDID(symtab,id,attrlist),id1) =
              IF id = id1
                  THEN TRUE
                  ELSE ISINBLOCK(symtab,id1),
      7)  RETRIEVE(INIT,id) = UNDEFINED,
      8)  RETRIEVE(ENTERBLOCK(symtab),id) = RETRIEVE(symtab,id),
      9)  RETRIEVE(ADDID(symtab,id,attrlist),id1) =
              IF id = id1
                  THEN attrlist
                  ELSE RETRIEVE(symtab,id1).
```

Figure 2.3: The Symboltable Data Type

As an aid to understanding these axioms, it is useful to consider a direct implementation. We let the representation be trees of INIT, ENTERBLOCK, and ADDID nodes and use the full set of semantic axioms as programs. Then, for example, if this direct implementation is used by a compiler in processing the following program segment,

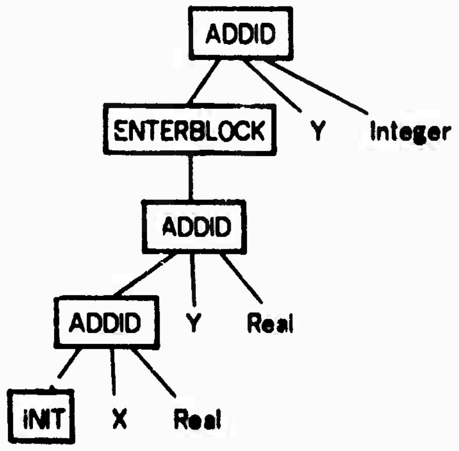
```
        begin
            real X,Y;
              ...
```

```
                begin
                        integer Y;
                          —
                end
        end
```

the symbol table

SYM=ADDID(ENTERBLOCK(ADDID(ADDID(INIT,X,Real),Y,Real)),Y,integer)

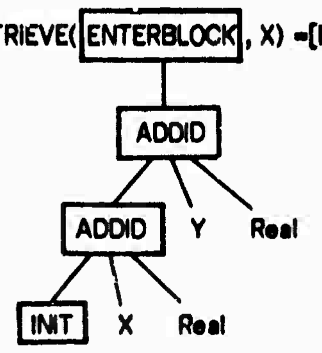
will be created within the innermost block. Diagrammed as a tree structure, this is

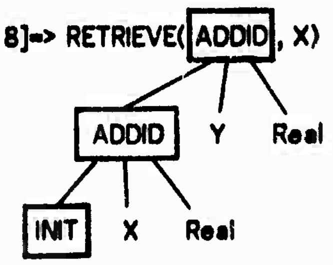


Suppose now that we apply RETRIEVE to SYM and X. Simulating the RETRIEVE operation using the direct implementation, we have

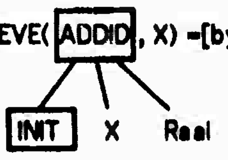
RETRIEVE(SYM,X)

=[by axiom 9]=> RETRIEVE(ENTERBLOCK , X) =[by axiom 8]=> RETRIEVE(ADDID , X)





=[by axiom 9]=> RETRIEVE(ADDID , X) =[by axiom 9]=> Real.



If tree structure operations are implemented with reasonable efficiency, then this direct implementation could actually be used in a compiler to test the Symboltable specification more extensively and, simultaneously, to test other components of the compiler. However, because this implementation requires potentially long searches for the RETRIEVE, ISINBLOCK, and LEAVEBLOCK operations, it is not very efficient. In the next section we turn to the problem of designing a more efficient implementation.

3. *ALGEBRAIC AXIOMS AS AN AID TO TOP-DOWN DEVELOPMENT OF IMPLEMENTATIONS*

The key to successful top-down design is the ability to construct, at each level of refinement, abstractions which suppress all irrelevant detail while clearly exposing the relevant concepts and structure. By deferring detail one reduces the number of decisions that must be made at any one time. Verifying the correctness of each refinement as it is developed is crucial. Therefore, the specification of a refinement, though possibly quite abstract, must be complete and unambiguous. All symbols that appear in it must be well-defined.

Though systems have occasionally been designed in a top-down fashion, they have for the most part been tested from the bottom up. This was necessary because the upper levels could not be easily tested in the absence of an implementation of lower levels. By eliminating (via direct implementations) the necessity of supplying such implementations, one eliminates the need to delay testing while awaiting the implementation of other modules. More importantly, if one executes with specifications rather than implementations of abstract operations, the possible sources of a known error are far more limited.

The ability to use specifications for testing is closely related to the policy of restricted information flow [Parnas72]. If a programmer is supplied with algebraic definitions of the abstract operations available to him and is forced to write and test his module with only that information available to him, he is denied the opportunity to rely intentionally or accidentally upon information that should not be relied upon. This not only serves to localize the effect of implementation errors, but also to increase the ease with which one implementation may be replaced by another. This should, in general, serve to limit the danger of choosing a poor representation and becoming inextricably locked into it.

In this section we carry out a design of a hierarchically structured implementation of the Symboltable data type, using algebraic specifications of the data types employed at each level of the implementation. Such a design was first presented in [Guttag75] and [Guttag76a], where the lowest level of the implementation was expressed as a set of PL/1-like programs. We differ here in that we continue to use the restricted set of language features described in Section 2. We further assume that direct implementations are efficiently implemented by tree structures, as discussed in Section 2 (and more fully in Section 5), so that some of the data types we use can be assumed to be directly implemented with reasonable efficiency. All of these assumptions are satisfied, or can easily be programmed, in real programming languages such as PL/1 or Pascal.

Let us now consider how we might proceed to design a reasonably efficient implementation of the Symboltable data type. First, note that if we ignored the complication introduced by block structure, a symbol table could be viewed abstractly as providing a mapping from identifiers to attribute lists. One way to handle block structure, especially suitable in a one-pass compiler, is to have a stack of mappings, each mapping from identifiers to attribute lists, with the top mapping on the stack corresponding to the current innermost block being processed. This is the method we have chosen in the implementation given in Figure 3.1.

```
representation

SYMT(Stack[Mapping[Identifier,Attributelist]]) → Symboltable.

programs

        declare stk:Stack, id:Identifier, attrlist:Attributelist;
        INIT = SYMT(PUSH(NEWSTACK,NEWMAP)),
        ENTERBLOCK(SYMT(stk)) = SYMT(PUSH(stk,NEWMAP)),
        ADDID(SYMT(stk),id,attrlist) =
                SYMT(REPLACE(stk,DEFMAP(TOP(stk),id,attrlist))),
        LEAVEBLOCK(SYMT(stk)) =
                IF ISNEW(POP(stk))
                        THEN SYMT(REPLACE(stk,NEWMAP))
                        ELSE SYMT(POP(stk)),
        ISINBLOCK(SYMT(stk),id) = ISDEFINED(TOP(stk),id),
        RETRIEVE(SYMT(stk),id) =
            IF ISNEW(stk)
                  THEN UNDEFINED
                  ELSE IF ISDEFINED(TOP(stk),id)
                        THEN EVMAP(TOP(stk),id)
                        ELSE RETRIEVE(SYMT(POP(stk)),id).
```

Figure 3.1: An Implementation of the Symboltable Data Type with a Stack of Mappings

This implementation uses the operations of the Stack data type schema of Figure 2.1 and the Mapping data type schema of Figure 3.2. Note that we have bound the parameters of the Stack and Mapping types. Given the syntactic specifications, complete type checking of the semantic specification is possible.

```
type Mapping[domaintype:Type,rangetype:Type]

syntax

        NEWMAP → Mapping,
        DEFMAP(Mapping,domaintype,rangetype) → Mapping,
        EVMAP(Mapping,domaintype) → rangetype,
        ISDEFINED(Mapping,domaintype) → Boolean.

semantics

        declare map:Mapping, dval,dval1:domaintype, rval:rangetype;
        EVMAP(NEWMAP,dval) = UNDEFINED,
        EVMAP(DEFMAP(map,dval,rval),dval1) =
                IF dval = dval1 THEN rval ELSE EVMAP(map,dval1),
        ISDEFINED(NEWMAP,dval1) = FALSE,
        ISDEFINED(DEFMAP(map,dval,rval),dval1) =
                IF dval = dval1 THEN TRUE ELSE ISDEFINED(map,dval1).
```

Figure 3.2: Mapping Data Type

The Mapping data type is fundamental. The Array data type of most programming languages is basically the same as the Mapping data type, except that usually no operation corresponding to ISDEFINED is provided. The specifications for an Array data type are given in Figure 3.3.

```
type Array[domaintype:Type,rangetype:Type]

syntax

        NEWARRAY → Array,
        ASSIGN(Array,domaintype,rangetype) → Array,
        ACCESS(Array,domaintype) → rangetype.

semantics

        declare arr:Array, dval,dval1:domaintype, rval:rangetype;
        ACCESS(NEWARRAY,dval) = UNDEFINED,
        ACCESS(ASSIGN(arr,dval,rval),dval1) =
                IF dval = dval1 THEN rval ELSE ACCESS(arr,dval1).
```

Figure 3.3: Array Data Type

The ASSIGN operation corresponds to DEFMAP and ACCESS to EVMAP. Note that multidimensional arrays can be dealt with by allowing domaintype to be the cross-product of several domains. In most programming languages the domain(s) of arrays must be a subset of the Integers or the cross-product of such subsets. In Figure 3.3, no such restriction is made, but in all of the uses of arrays which follow, the domain type will be Integer.

Before continuing to refine these operations, i.e., before supplying implementations for types Stack and Mapping, we should consider the problem of ascertaining whether or not the above implementation of the Symboltable data type is correct. This can be approached formally using the proof techniques to be described in Section 4, or less formally using testing techniques based on direct implementation of types Stack and Mapping. Had we, for example, tested portions of the compiler by using a direct implementation of type Symboltable (Figure 2.3), we might now run the same tests using the current implementation of type Symboltable and direct implementations of types Stack and Mapping. We shall not further discuss such testing here, but in Section 4 we shall carry out parts of the formal proof of correctness.

Earlier we noted that the development process could be halted when one arrived at a program all of whose operations were either primitive or efficiently directly implementable. We have not yet reached that point. While the use of the Stack yields a significant improvement in the time required for the LEAVEBLOCK operation (over the time required by the direct implementation of the Symboltable type), the time required for the RETRIEVE operation has not been improved at all. For completeness we therefore proceed to the problem of efficiently implementing the Mapping type. The remaining sections do not rely upon this material, thus the reader may want to skip directly to the start of Section 4.

The Mapping data type has many implementations which would be reasonably efficient for the Symboltable application, e.g., using balanced tree structures or hash tables. Figure 3.4 shows an implementation using a hash table. Note that the parameters domaintype and rangetype have been bound to identifier and Attributelist. A hash table can be viewed abstractly as a composition of mappings, of which one is implementable by "random access" techniques to provide efficiency in searching. In Figure 3.4 we have assumed that Arrays with integer domains have random access characteristics. We have left the other mappings abstract; i.e., Mapping1 is a data type identical to Mapping, except for renaming. For convenience we have assumed an operation for initializing an Array:

```
        INITIALIZE(Array[Integer,Attributelist],Integerrange,Attributelist) →
                    Array[Integer,Attributelist]
```

with the axiom

```
        ACCESS(INITIALIZE(arr,irange,attrlist),int)
              = IF ISIN(int,irange)
                        THEN attrlist
                        ELSE ACCESS(arr,int)
```

Thus INITIALIZE is a generalization of the ASSIGN operator. In most programming languages, no INITIALIZE operator for arrays is provided, but, of course, it is easily implemented with a loop over the elements of irange.

Regarding the HASH operation, we assume only that its syntactic specification is

HASH(Identifier) → Hashrange

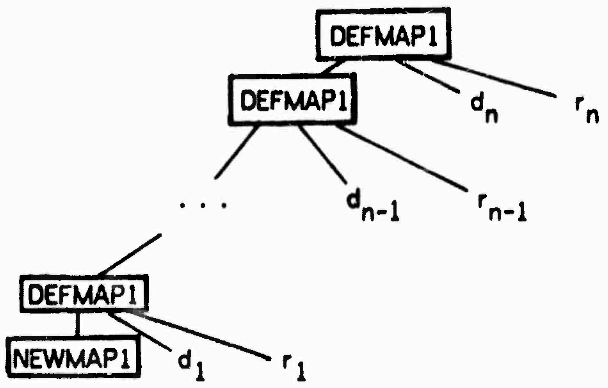
for some particular range Hashrange of Integer values. This is sufficient to show the correctness of the implementation of Figure 3.4; the distribution of identifiers over the range is of concern only as a matter of efficiency.

```
representation   MAPP(Array[Integer,Mapping1[Identifier,Attributelist]])
                                          → Mapping[Identifier,Attributelist]

programs

     declare arr:Array, id:Identifier, attrlist:Attributelist;
     NEWMAP = MAPP(INITIALIZE(NEWARRAY,Hashrange,NEWMAP1)),
     DEFMAP(MAPP(arr),id,attrlist) =
               MAPP(ASSIGN(arr,HASH(id),DEFMAP1(ACCESS(arr,HASH(id)),id,attrlist))),
     EVMAP(MAPP(arr),id) = EVMAP1(ACCESS(arr,HASH(id)),id),
     ISDEFINED(MAPP(arr),id) = ISDEFINED1(ACCESS(arr,HASH(id)),id).
```

Figure 3.4: Implementation of Mapping Data Type with a Hash Table

For the implementation of the Stack data type we could use the implementation with Array/Integer records, as already given in Figure 2.2. Another possibility is the direct implementation using tree structures composed of PUSH and NEWSTACK nodes, as discussed in Section 2.2.

We will use a direct implementation of the Mapping1 data type, representing a Mapping1 value with a tree structure of the form



where the d_i are domain values and the r_i are range values.

In a complete implementation the Identifier and Attributelist data types also must be dealt with, but we shall ignore them here, as we do not regard them as being part of the Symboltable data type. (The Symboltable implementation as we have given it requires very little interaction with the Identifier data type--only the HASH and Identifier equality operations interact--and essentially no interaction with the Attributelist data type.) Thus, under the assumptions discussed at the beginning of this section, the design of the implementation is complete. A diagram of the structure of the design is given in in Figure 3.5.

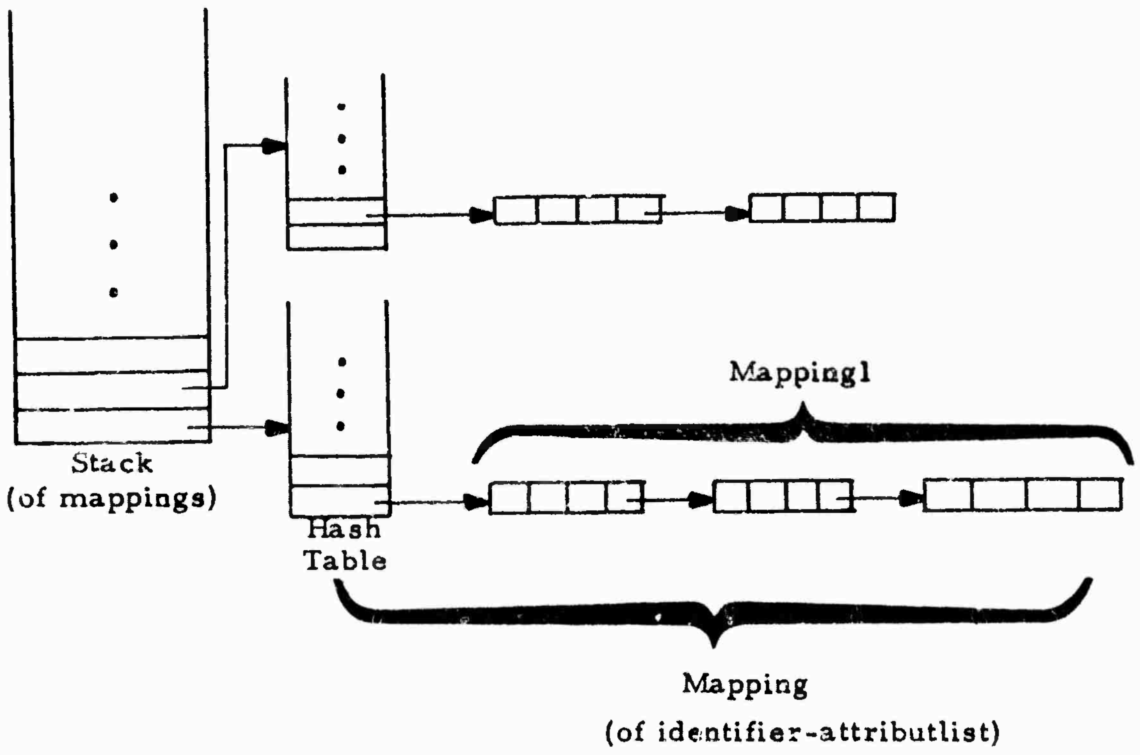


Figure 3.5: Symboltable Implementation via Hash Tables

The hierarchical relationship of the types used in this implementation is reflected in Figure 3.6.

$$\begin{tikzcd}
 & \text{Symboltable} \arrow[dl] \arrow[dr] & & \\
\text{Stack} \arrow[dr] \arrow[ddr] & & \text{Mapping} \arrow[dl] \arrow[ddl] \arrow[dr] & \\
 & \text{Array} & & \text{Mapping1} \\
 & \text{Integer} & &
\end{tikzcd}$$

Figure 3.6: Hierarchy of Types in Symboltable Implementation

4. *PROVING CORRECTNESS*

In this section we turn to the problem of proving correctness of implementations of data types. We shall continue to center the discussion upon the example of the Symboltable data type. We shall show parts of the proof of correctness of the example implementation given in Section 3.

One of the most important aspects of the proof techniques used to prove correctness of algebraically specified data types is that the proofs can be factored into levels corresponding to the implementation levels. To prove the correctness of a data type implemented in terms of other data types, we need to rely only on the axiomatic specifications of the other data types, *not* on their implementations. For example, as we will show in Section 4.2, to verify the top level of the implementation of the Symboltable data type, we use the semantic axioms for the Stack and Mapping data types, and ignore their implementations. In fact, we could (and, in general, would) have proceeded with the proof of the top level even before implementing the Stack or Mapping data types.

Another highly significant aspect of the use of axioms for other data types in the proof of an implementation is the computational nature of the proof steps: the axioms are used as rewrite rules and proofs proceed via a series of reductions. This aspect is also illustrated in Section 4.2. The main implication of this is that to a large extent the proof process can be easily automated. In Section 5 we shall discuss some of the details of an interactive system for verifying implementations of data types which is in use at the USC Information Sciences Institute.

Two extremely important considerations in verifications of data type implementations are *data type invariants* and *equality interpretations*. Again, the algebraic axiom approach seems particularly well suited to dealing with these considerations, as will be shown in Sections 4.3 and 4.4. Section 4.5 contains a summary of the proof procedure and a discussion of a technical difference between our approach to equality and that taken by other authors.

4.1 *FORMAL DEDUCTION USING THE BOOLEAN DATA TYPE*

Although we will not be completely formal in the example proofs in the following sections, it is useful to discuss at this point the basis we have chosen for automating such proofs, since it fits very well with the overall approach of using algebraically specified data types. In fact, many of the formal deductions will be based on the following algebraic specification of the Boolean data type:

type Boolean

syntax

TRUE → Boolean,
FALSE → Boolean,
(IF Boolean THEN Boolean ELSE Boolean) → Boolean,
(Boolean ∧ Boolean) → Boolean,
(Boolean OR Boolean) → Boolean,
¬Boolean → Boolean,
(Boolean ⊃ Boolean) → Boolean,
(Boolean ≡ Boolean) → Boolean.

semantics

declare p,q,r:Boolean;
(IF TRUE THEN q ELSE r) = q,
(IF FALSE THEN q ELSE r) = r,
(p ∧ q) = IF p THEN q ELSE FALSE,
(p OR q) = IF p THEN TRUE ELSE q,
¬p = IF p THEN FALSE ELSE TRUE,
(p ⊃ q) = IF p THEN q ELSE TRUE,
(p ≡ q) = IF p THEN q ELSE ¬q,

Figure 4.1: Boolean Data Type

In this specification each of the usual logical operators is related by an axiom to the IF-THEN-ELSE operator, which is axiomatized by the first two axioms relating it to TRUE and FALSE. The use of ∧, OR, etc. in infix rather than prefix form is purely for convenience. Although the above specification defines IF-THEN-ELSE only for Boolean operands, we assume that every data type T can use IF-THEN-ELSE with the syntactic specification

(IF Boolean THEN T ELSE T) → T

and the same axioms as the first two in the above specification.

We shall make frequent use of the following rewrite rules for IF-THEN-ELSE, which are theorems provable from the Boolean axioms:

1. [Repeated result rule] (IF p THEN q ELSE q) = q

2. [Redundant IF rule] (IF p THEN TRUE ELSE FALSE) = p

3. [IF-distribution rule] (IF(IF p THEN q ELSE r) THEN a ELSE b) =
IF p THEN (IF q THEN a ELSE b) ELSE (IF r THEN a ELSE b)

4. [Logical substitution rules]
   (IF p THEN q[p] ELSE r) = (IF p THEN q[TRUE *for* p] ELSE r)
   (IF p THEN q ELSE r[p]) = (IF p THEN q ELSE r[FALSE *for* p]).

In the left-hand side of a rule, "q[p]" means q must be an expression in which p occurs as a subexpression (possibly p=q). In the right hand side, "q[TRUE *for* p]" is the result of substituting TRUE for all occurrences of p in q. We require that p occur as a subexpression of q to limit applicability of the rule to those places where it will effect a change. Rules 1-4 are also theorems for IF-THEN-ELSE in other data types.

As noted in [McCarthy63] and [Boyer75], the Boolean axioms and rules 1-4 combine to yield a system for "simplifying" expressions of the propositional calculus, such as

a) $A \wedge TRUE$,
b) (A OR B) OR A,
c) $((A \wedge B) \wedge A) \supset (B \wedge C)$
d) $((A \text{ OR } B) \supset C) \equiv ((A \supset C) \wedge (B \supset C))$

For example,

```
((A OR B) OR A) ==[by OR axiom]==>
      (IF (IF A THEN TRUE ELSE B) THEN TRUE ELSE A)

==[by IF-distribution rule]==>
      (IF A THEN (IF TRUE THEN TRUE ELSE A)
             ELSE (IF B THEN TRUE ELSE A))

==[by IF axiom]==>
      (IF A THEN TRUE ELSE (IF B THEN TRUE ELSE A))

==[by logical substitution rule]==>
      (IF A THEN TRUE ELSE (IF B THEN TRUE ELSE FALSE))

==[by Redundant IF rule]==> (IF A THEN TRUE ELSE B)
```

The result is equivalent to (A OR B). Similarly, a) can be reduced to A, c) can be reduced to a form equivalent to $(A \wedge B) \supset C$, and d) can be reduced to TRUE.

In Section 5 we shall in fact prove that the Boolean axioms and rules 1-4 are *complete* with respect to the propositional calculus, in that any valid expression in propositional calculus (i.e., provable by truth table) is reducible to TRUE by use of these rewrite rules. These rules form the basis for an automatic simplifier for logical expressions to be described in Section 5.

It is, of course, necessary to go beyond propositional calculus and include deductive rules for equality and other operators. We will not go in to the details of the formal rules here, except to mention the following important rule:

5. [Case analysis rule] $f(a_1, \ldots, (\text{IF } p \text{ THEN } x \text{ ELSE } y), \ldots, a_n) =$
$\text{IF } p \text{ THEN } f(a_1, \ldots, x, \ldots, a_n)$
$\text{ELSE } f(a_1, \ldots, y, \ldots, a_n)$
when $f \neq$ IF-THEN-ELSE.

This is a "second-order" rewrite rule and the rule applies to IF-THEN-ELSE expressions in any operand position. The case in which f is IF-THEN-ELSE is already covered in part by the IF-distribution rule. Note that if f were permitted to be IF-THEN-ELSE, then the expression IF a_1 THEN (IF p THEN x ELSE y) ELSE a_3 could be transformed to IF p THEN (IF a_1 THEN x ELSE a_3) ELSE (IF a_1 THEN y ELSE a_3) and the rule would apply again, leading to an infinite sequence of applications. An important application of this case analysis rule occurs when f is "=", e.g.,

((IF p THEN x ELSE y) = z) ==[by Case analysis rule]==> (IF p THEN (x=z) ELSE (y=z)).

4.2 *VERIFICATION OF ONE OF THE SYMBOLTABLE AXIOMS*

The basic proof technique for verifying an implementation of a data type is to show that each of the axiomatic specifications for the data type is satisfied when the programs for the operations of the data type are substituted into the axioms. As our first example, consider the ninth axiom for the Symboltable data type:

```
RETRIEVE(ADDID(symtab,id,attrlist),id1) =
        IF id = id1 THEN attrlist
                    ELSE RETRIEVE(symtab,id1).                  (4.2-1)
```

To start, we assume there exists a stack stk such that symtab = SYMT(stk). (We will show in Section 4.3 how to verify this assumption.) Substituting, we obtain the verification condition

```
RETRIEVE(ADDID(SYMT(stk),id,attrlist),id1) =
        IF id = id1 THEN attrlist
                    ELSE RETRIEVE(SYMT(stk),id1).               (4.2-2)
```

The goal now is to show that this equation is true using the programs for RETRIEVE and ADDID (Figure 3.1) and the axioms for the Stack and Mapping data types (Figures 2.1 and 3.2) as rewrite rules. We must also use the axioms and some theorems for the Boolean data type, as discussed in Section 4.1.

Working first on the left-hand side of (4.2-2), we make the following reductions:

```
LHS ==[by ADDID program]==>
    RETRIEVE(SYMT(stk1),id1) where stk1 = REPLACE(stk,DEFMAP(TOP(stk),id,attrlist))

==[by RETRIEVE program]==>
 IF ISNEW(stk1)
     THEN UNDEFINED
     ELSE IF ISDEFINED(TOP(stk1),id1)
             THEN EVMAP(TOP(stk1),id1)
             ELSE RETRIEVE(SYMT(POP(stk1)),id1)
```

```
==[by REPLACE, ISNEW, POP and TOP axioms]==>
   IF ISDEFINED(DEFMAP(TOP(stk),id,attrlist),id1)
                THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by ISDEFINED axiom]==>
   IF (IF id = id1 THEN TRUE ELSE ISDEFINED(TOP(stk),id1))
                THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF-distribution Rule]==>
   IF id = id1
     THEN IF TRUE
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)
     ELSE IF ISDEFINED(TOP(stk),id1)
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF axiom]==>
   IF id = id1
     THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
     ELSE IF ISDEFINED(TOP(stk),id1)
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by EVMAP axiom]==>
   IF id = id1
        THEN (IF id = id1 THEN attrlist ELSE EVMAP(TOP(stk),id1))
        ELSE IF ISDEFINED(TOP(stk),id1)
                THEN (IF id = id1 THEN attrlist ELSE EVMAP(TOP(stk),id1))
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by Logical substitution rule]==>
   IF id = id1
        THEN (IF TRUE THEN attrlist ELSE EVMAP(TOP(stk),id1))
        ELSE IF ISDEFINED(TOP(stk),id1)
                THEN (IF FALSE THEN attrlist ELSE EVMAP(TOP(stk),id1))
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF axioms]==>
   IF id = id1
        THEN attrlist
        ELSE IF ISDEFINED(TOP(stk),id1)
                THEN EVMAP(TOP(stk),id1)
                ELSE RETRIEVE(SYMT(POP(stk)),id1)
```

Having already substituted once for RETRIEVE, we do not substitute again for the recursive call. The right hand side of (4.2-2) reduces to the same expression upon use of the program for RETRIEVE. Thus (4.2-2) has been shown to be true, and axiom (4.2-1) has been verified for the implementation.

The reason for showing the proof of the axiom in this much detail is that we wish to make clear that each step of the proof is a straightforward application of a reduction rule. The only choice to be made in terms of "proof strategy" is which of several possible reductions to apply at the next step. It should be noted, however, that the use of the axioms as left-to-right rewrite rules can be combined with restrictions on the form of the axioms (to be discussed in Section 5) to preclude "infinite loops" in the proof process. Thus, if we assume that the axioms are satisfiable, all possible reduction sequences will terminate with the same result [Rosen73]; the reduction process is, therefore, readily automated. The above proof was carried out fully automatically by the prototype data type verification system at USC Information Sciences Institute.

Verification of the axioms in the manner of the above proof actually establishes only *partial correctness* of the implementation, i.e., that if the programs terminate they give results satisfying the axioms. Proof of termination must be done separately. However, implementations of data types are often simple enough that termination is obvious, and we shall not deal with the issue of formal proofs of termination in this report.

4.3 *USE AND VERIFICATION OF DATA TYPE INVARIANTS*

Although the lengthy example axiom verification of the previous section required the use of many different rewrite rules, it did not illustrate the use of ***data type invariants***. Consider, for example, the second symbol table axiom:

$$\text{LEAVEBLOCK(ENTERBLOCK(symtab))} = \text{symtab.}$$

Substituting

$$\text{symtab} = \text{SYMT(stk)} \qquad (4.3\text{-}1)$$

and applying all possible reductions (including the case analysis rule with f = "="), we obtain

$$\begin{aligned}&\text{IF ISNEW(stk) THEN (SYMT(PUSH(NEWSTACK,NEWMAP))} = \text{SYMT(stk))}\\&\qquad\text{ELSE (SYMT(stk)} = \text{SYMT(stk)).}\end{aligned} \qquad (4.3\text{-}2)$$

We cannot reduce this equation to TRUE unless we know that

$$\text{ISNEW(stk)} = \text{FALSE.} \qquad (4.3\text{-}3)$$

To prove (4.3-3), we recall that stk is not just an arbitrarily chosen stack, but one assumed to be generated as a representation of a symbol table. If we examine the syntactic specification of the Symboltable data type, we see that the only operations which produce symbol tables as their output are INIT, ENTERBLOCK, ADDID, and LEAVEBLOCK. Examining the program for each of these operations, we see that INIT generates an initial stack, stk, for which (4.3-3) is true, and that if (4.3-3) is true of the stack representing the symbol table argument of any of the other operations, then it is true of the stack produced in the result. Therefore, (4.3-3) must be true of all stacks produced as representations of symbol tables by operations of the Symboltable data type.

The general principle being used in the above proof is that of *data type induction* (called "generator induction" in [Spitzen75] and [Wegbreit76]). Paraphrasing the discussion in [Spitzen75],* we suppose that a data type T has, according to its syntactic specification, exactly the operations $F_1, \ldots, F_t$ whose range is the set of values of T. Let P(x) be a property of values of type T. Then if the truth of P for arguments of type T of each F_i implies the truth of P for the results of calls of F_i allowed by the syntactic specification of T, then it follows that P is true of all values of the data type. If strong type-checking is assumed, the validity of this rule follows by induction on the number of computation steps involving values of type T. As Spitzen and Wegbreit point out, the data type induction principle "is analagous to the principle of complete induction over the integers. As with complete induction, one of the results which must be established is the base step, that P is true of the results of those primitives F with no arguments of type T." In the case of symbol tables, INIT is the only such primitive.

Let us examine more carefully the proof of (4.3-3) by data type induction. We can regard a property P_1 that we wish to prove about symbol tables by data type Induction as an operation on symbol tables with the syntactic specification

P_1(Symboltable) → Boolean,

and the semantic specifications

```
P_1(INIT)
P_1(symtab) ⊃ P_1(ENTERBLOCK(symtab))
P_1(symtab) ⊃ P_1(ADDID(symtab,id,attrlist)
P_1(symtab) ⊃ P_1(LEAVEBLOCK(symtab))                                  (4.3-4)
```

which can be generated automatically from the syntactic specification of the Symboltable data type. To prove (4.3-3), we let the interpretation of P_1 in terms of the implementation values be

P_1(SYMT(stk)) = (ISNEW(stk) = FALSE). (4.3-5)

We then prove each of the conditions in (4.3-4) under this interpretation of P_1 along with the implementation programs of the Symboltable operations. For example, using (4.3-1), the fourth condition becomes

```
P_1(SYMT(stk)) ⊃ P_1(LEAVEBLOCK(SYMT(stk)))

==[by (4.3-5)and LEAVEBLOCK program]==>
(ISNEW(stk) = FALSE)
    ⊃ P_1(IF ISNEW(POP(stk)) THEN SYMT(REPLACE(stk,NEWMAP))
                                    ELSE SYMT(POP(stk)))
```

----

* Page 141

```
==[by case analysis rule]==>
(ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk)) THEN P_1(SYMT(REPLACE(stk,NEWMAP)))
                                    ELSE P_1(SYMT(POP(stk))))

==[by (4.3-5)]==>
(ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk)) THEN ISNEW(REPLACE(stk,NEWMAP)) = FALSE
                                    ELSE ISNEW(POP(stk)) = FALSE)

==[by ISNEW axiom and logical substitution rule]==>
(ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk)) THEN FALSE = FALSE
                                    ELSE FALSE = FALSE
```

which reduces to TRUE with the application of the reflexive property of Boolean equality, the repeated result rule, and the ⊃ axiom.

A property P which is true of all values of a data type is called an ***invariant of the data type***. If the proof requires interpretation of P in terms of the implementation, then it is called an ***implementation invariant*** (of the data type). Thus (with the establishment of the other three conditions in (4.3-4)), we have almost completed showing that (4.3-3) is an implementation invariant of the Symboltable data type. What remains to be shown is that

$$P(symtab) = (\exists\ stk \in Stack,\ symtab = SYMT(stk)) \qquad (4.3\text{-}6)$$

is an implementation invariant. Again, this can be easily verified using data type induction. The invariant (4.3-6) is an example of a ***representation invariant***, which we define to be that implementation invariant which describes how the abstract values are represented. The representation invariant can be constructed automatically from the ***representation*** part of the implementation.

As an example of an invariant of a data type other than an implementation invariant, consider

```
P(symtab) = (symtab = INIT)
                OR (∃ symtab1, symtab = ENTERBLOCK(symtab1))
                OR (∃ symtab1,id,attrlist,
                        symtab = ADDID(symtab1,id,attrlist)).          (4.3-7)
```

which can be shown to satisfy each of the conditions in (4.3-4), using the syntactic specification and semantic axioms for the Symboltable data type. Invariants such as (4.3-7) are useful in proofs which make use of the data type, in that they can be used to reduce to a minimum the number of cases that must be considered in a proof by case analysis. Note that (4.3-7) can also be regarded as the representation invariant for the direct implementation of the Symboltable data type, as discussed in Section 2.3.

4.4 *INTERPRETATION OF THE EQUALITY OPERATOR*

Another important consideration in proving that an implementation satisfies an axiom is the interpretation of the equal sign in the axiom in terms of the implementation. To illustrate this, consider the axiom for stacks, POP(PUSH(stk,elm)) = stk, and the implementation of Figure 2.2. We assume there exists an array arr and an integer t such that stk = STAK(arr,t). Substituting this into the axiom, we obtain

```
      POP(PUSH(STAK(arr,t),elm)) = STAK(arr,t)

  ==[by PUSH program]==>
      POP(STAK(ASSIGN(arr,t+1,elm),t+1)) = STAK(arr,t)

  ==[by POP program and Integer theorem ((t+1)-1 = t)]==>
      (IF t+1 = 0 THEN STAK(ASSIGN(arr,t+1,elm),0)
             ELSE STAK(ASSIGN(arr,t+1,elm),t)) = STAK(arr,t).
```

Assuming $t \geq 0$, which can be proved as an implementation invariant by the methods of Section 4.3, we can make a further reduction, so that the equation becomes

STAK(ASSIGN(arr,t+1,elm),t) = STAK(arr,t). (4.4-1)

What we now need to prove the validity of (4.4-1) is a proof that STAK(ASSIGN(arr,t+1,elm),t) and STAK(arr,t) are indistinguishable by any sequence of operations mapping us out of type Stack. To do this we introduce an *equality interpretation* for this representation of type Stack. To the implementation of Figure 2.2 we thus add

```
equality interpretation
      (STAK(arr,t) = STAK(arr1,t1)) =
       (t = t1) ∧ ∀k(1≤k≤t ⊃ ACCESS(arr,k) = ACCESS(arr1,k))        (4.4-2)
```

Using (4.4-2), (4.4-1) becomes:

```
       (t = t) ∧ ∀k(1≤k≤t ⊃ ACCESS(ASSIGN(arr,t+1,elm),k) = ACCESS(arr,k))

    ==[by equality axiom for integers and ACCESS axiom for arrays]==>
       ∀k(1≤k≤t ⊃ (IF t+1=k THEN elm ELSE ACCESS(arr,k)) = ACCESS(arr,k))

    ==[by inequality property of integers]==>
       ∀k(1≤k≤t ⊃ (IF FALSE THEN elm ELSE ACCESS(arr,k)) = ACCESS(arr,k))
```

which is reduced to TRUE by use of the IF and ⊃ axioms, the repeated result rule, and the property ∀k(TRUE) = TRUE.

We must also prove that the chosen interpretation of equality of stacks has the properties of an equality operator, as part of the verification of the implementation. These properties are

1. (Reflexivity) x=x
2. (Symmetry) x=y ⊃ y=x

3. (Transitivity) $x=y \wedge y=z \supset x=z$
4. (Substitution) $x=y \supset p = p[y$ *for* $x]$ where p is any expression.

Properties 1-3 are easily proved for the equality interpretation (4.4-2) for stacks, since they reduce to the corresponding properties for equality in type elementtype. To prove the substitution property it suffices to show that the implementation satisfies

4a. stk=stk1 ⊃ PUSH(stk,elm)=PUSH(stk1,elm)
4b. stk=stk1 ⊃ POP(stk)=POP(stk1)
4c. stk=stk1 ⊃ TOP(stk)=TOP(stk1)
4d. stk=stk1 ⊃ ISNEW(stk)=ISNEW(stk1)
4e. stk=stk1 ⊃ REPLACE(stk,elm)=REPLACE(stk1,elm)

since, by the syntactic specification, Stack values can appear in expressions only as the arguments of PUSH, POP, TOP, ISNEW, or REPLACE operations. The proof of 4a is, letting stk=STAK(a,t), stk1=STAK(b,u),

```
STAK(a,t)=STAK(b,u) ⊃ PUSH(STAK(a,t),elm)=PUSH(STAK(b,u),elm)

==[by PUSH program and Stack equality interpretation]==>
[((t=u) ∧ ∀k(1≤k≤t ⊃ ACCESS(a,k)=ACCESS(b,k))]
              ⊃ [((t+1=u+1) ∧ ∀k(1≤k≤t+1
                              ⊃ ACCESS(ASSIGN(a,t+1,elm),k)=
                                        ACCESS(ASSIGN(b,u+1,elm),k))]

==[by substitution of u for t]==>
              [((t=u) ∧ ∀k(1≤k≤u ⊃ ACCESS(a,k)=ACCESS(b,k))]
               ⊃ ∀k(1≤k≤u+1 ⊃ ACCESS(ASSIGN(a,u+1,elm),k)=
                                            ACCESS(ASSIGN(b,u+1,elm),k))
```

The equation between ACCESS expressions reduces to IF u+1=k THEN elm=elm ELSE ACCESS(a,k)=ACCESS(b,k), and thus the conclusion can be reduced to match the second hypothesis. We omit the details of the proofs of 4a-4e, but note that they have been proved automatically by the programs described in Section 5.

In the general case of a data type T with operations $F_1, \ldots, F_n$, the verification conditions for the substitution property are

$$x=y \supset F_i(\ldots,x,\ldots) = F_i(\ldots,y,\ldots), \qquad i=1,\ldots,n$$

where the other arguments are held fixed.

For some data types, the equality operation will be among the operations listed in the syntactic specification, which we will interpret to mean that it must be implemented by a program. In such a case, the program defines an equality interpretation. Nevertheless, it may still be useful to give another equality interpretation for use in verifying the implementation. For example, Stack equality might be programmed as

```
(STAK(a,t)=STAK(b,u)) =
        IF t=0
             THEN (u=0)
             ELSE ((t=u) ∧ (ACCESS(a,t)=ACCESS(b,t))
                  ∧ (STAK(a,t-1)=STAK(b,t-1))).
```

While we could use this recursive definition in the proofs instead of (4.4-2), it turns out that the proofs are somewhat easier if (4.4-2) is used.

Our use of equality interpretations is a generalization of an earlier method using *abstraction functions* [Hoare72], [Guttag75]. An abstraction function is a function A(x) which maps representation values x into the abstract values which they represent. As an example, an abstraction function for the implementation of the Stack data type by Array/Integer pairs can be defined by

```
S(arr,t) = IF t=0 THEN NEWSTACK
              ELSE PUSH(S(arr,t-1),ACCESS(arr,t)).
```

Given an abstraction function, an equality interpretation can be defined in terms of it, e.g.

```
(STAK(a,t)=STAK(b,u)) = (S(a,t)=S(b,u));
```

but the opposite is not true; this is why the use of equality interpretations is a generalization of the use of abstraction functions.

4.5 *SUMMARY AND COMPARISON TO ANOTHER PROOF TECHNIQUE*

Let us now summarize the main steps of the proof procedure for verifying data type implementations using algebraic specifications. We suppose that we have an algebraic specification of a data type and an implementation expressed in terms of other data types for which we also have algebraic specifications. Then the main steps are as follows:

1. State the *representation invariant* of the implementation and prove it using data type induction (as explained at the end of Section 4.3).

2. State the *equality interpretation* of the implementation as discussed in Section 4.4.

3. Using the representation invariant, substitute the representation into the axioms of the data type and into the equality axioms (reflexive, symmetric, transitive, substitution), obtaining a set of verification conditions (see Section 4.2).

4. Prove each of the verification conditions, using as rewrite rules the programs of the implementation, the equality interpretation, and the axioms of the data types used in the programs (including the Boolean, Integer, etc. data types). In some cases, completion of a proof will require one or more assumptions to be made about the representation or the data types used in the implementation (see Section 4.2).

5. Prove that the assumptions made in step 4, or a stronger set of assumptions, are invariants, using data type induction (Section 4.3).

In Section 5 we will discuss an interactive program which guides the user through steps 1, 3, 4, and 5, accomplishing many of the tasks automatically.

We conclude this section with a brief discussion of a technical difference between our proof technique and that of [Zilles75] and [Goguen74]. Regarding equality of the values of an abstract data type, we recall our basic assumption that the properties of the values can be derived solely from the relations determined by the axioms (Section 2.1). On the basis of this assumption, we conclude that two values may be assumed to be the same unless provably different. [Zilles75] and [Goguen74] make the opposite assumption, i.e., that values should be considered to be different unless they are demonstrably equal.

This difference in viewpoint is formally expressed in terms of the congruence relations defined by the axioms of the type. "The congruence relations used are the smallest congruence relations which contain all of the defining relations [axioms]. This means that two expressions are equivalent if and only if there is a sequence of expressions such that the first and last expressions are the expressions in question and every adjacent pair of expressions can be shown to be equivalent using some defining relation." [Zilles75] We, on the other hand, permit any congruence relations. An example may help to clarify this distinction.

Consider the abstract type Symboltable as defined in Figure 2.3 and examine two possible Symboltable values:

S = ADDID(ADDID(INIT,Y,Integer),X,Real)
T = ADDID(ADDID(INIT,X,Real),Y,Integer)

Using the axioms it cannot be shown that S = T; hence, under the smallest congruence viewpoint these values would be considered unequal. We prefer to regard S and T as equal, ignoring the difference in the order in which the identifiers are entered.

The practical ramifications of this difference in viewpoint are important. If one interprets the axioms as defining the smallest congruence relations, then the algebra defined by the axioms is unique up to isomorphism. All acceptable implementations of the abstract type must, therefore, yield algebras that are isomorphic to one another and to the original algebra. This observation is the basis of the technique used by both [Zilles75] and [Goguen74] to show the correctness of implementations of abstract data types.

This is a very restrictive notion of correctness. It requires that abstract values that are not provably equal be mapped onto distinct concrete values. That this is a significant restriction may be easily shown through reference to our example. One efficient implementation of the Symboltable data type was achieved by using a hashing function as in Figure 3.5. However, this implementation does not preserve the information necessary to distinguish between the expression S and T; it is therefore not a correct implementation with respect to the smallest congruence definition of correctness.

It should be pointed out that this restrictive notion of correctness does have certain advantages. The requirement that no information be lost increases the likelihood of being able to add new operations to the type without having to make extensive changes to existing implementations. Implementations which conform to our notion of correctness do not, in general, exhibit this property.

It should also be pointed out that by simply adding the axiom

ADDID(ADDID(symtab,id,attrs),id1,attrs1) = ADDID(ADDID(symtab,id1,attrs1),id,attrs),

our hashtable implementation becomes acceptable under the smallest congruence viewpoint. Axioms of that form, however, can lead to nonterminating computations when used as rewrite rules.

5. *AUTOMATIC TOOLS*

The previous sections have presented a methodology for the specification of data types and use of such specifications in design, testing, and verification. We have demonstrated with a number of examples that the methodology is reasonably straightforward to apply; indeed, many of the steps are "automatic" in the sense that there is no choice about what to do next. This suggests that it can be very profitably implemented by automatic tools whose use will speed up the application of the methods and avoid the errors which would inevitably occur amidst the tedium of applying such straightforward steps by hand. While we would argue that the results of using the methodology are valuable enough that it should be used even if it must be done by hand, we hope to show in this section that useful automatic tools can be achieved with a very modest investment of software development. We therefore feel that it is reasonable to advocate that such tools be incorporated into a wide variety of programming environments.

We first discuss in Section 5.1 the concepts of *direct implementations, expression data types*, and *reduction systems*. While direct implementations are useful for testing and in some cases can serve as actual implementations, reduction systems are useful for carrying out proofs of correctness of applications of abstract data types. In Section 5.2 we discuss the realization of reduction systems and direct implementations with a simple *pattern-match compiler*, based mainly on ideas from [Jenks74].

Reduction systems and direct implementations are in fact available essentially "for free" in symbolic mathematical systems, such as Reduce [Hearn71], Scratchpad [Griesmer71], [Jenks74], or Macsyma [Martin71], which include pattern-matching capabilities. (Such systems appear better suited to the realization of reduction systems than a seemingly more obvious choice, SNOBOL, since their pattern-matching capabilities are built around the operator-operand structure of mathematical expressions, whereas SNOBOL is character-oriented.) Our initial experience with data type reduction systems was in fact with Reduce. Since Reduce performs pattern matching interpretively, however, it is less efficient than use of the pattern-match compiler to be described in Section 5.2.

The programs which we have developed for use in carrying out proofs of correctness are currently being run as part of an interactive program verification system [Good75]. This is a large system of programs written in Lisp and Reduce, running on PDP-10 hardware. It includes a relatively sophisticated natural deduction theorem prover. We have, however, found little need for this facility in performing data type verifications. Most of the proofs are carried out via a new component called CEVAL ("Conditional Evaluator"), which is described in Section 5.3. The other main new component is DTVS ("Data Type Verification System"), a collection of user-interface programs. A brief description of DTVS will be given in Section 5.4.

5.1 *DIRECT IMPLEMENTATIONS, EXPRESSION DATA TYPES, AND REDUCTION SYSTEMS*

An important point about direct implementations is that they do not require any additions to the simple language for specifications used in Section 2. In fact, we define a *direct implementation* of a data type T to be an implementation whose representation part is a subset of the syntactic specification of T and whose program part is a subset of the semantic specification of T. The *maximal direct implementation* of a data type T is the implementation whose representation part is the entire syntactic specification of T and whose program part is the entire semantic specification for T.

Direct implementations need not deal with terms containing free variables. In performing proofs of correctness, on the other hand, we must deal with variables whose values are not known. If we add to the syntactic specification of the Symboltable data type (Figure 2.3) the operation

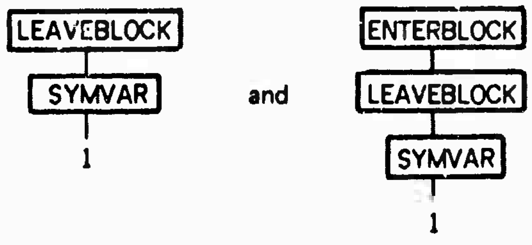
$$\text{SYMVAR(Integer)} \rightarrow \text{Symboltable}$$

but add no axioms relating this new operation to the other operations, we obtain a different data type which can be regarded as a *Symboltable Expression data type*. This is because SYMVAR(1), SYMVAR(2), etc., can be regarded as *variables* of type Symboltable, and because the lack of axioms relating SYMVAR to other operations makes it necessary to include expressions such as

LEAVEBLOCK(SYMVAR(1)),

ENTERBLOCK(LEAVEBLOCK(SYMVAR(1)))

among the values of the data type. In terms of the tree structures introduced in Section 2 to explain direct implementations, the values of a direct implementation should include trees such as

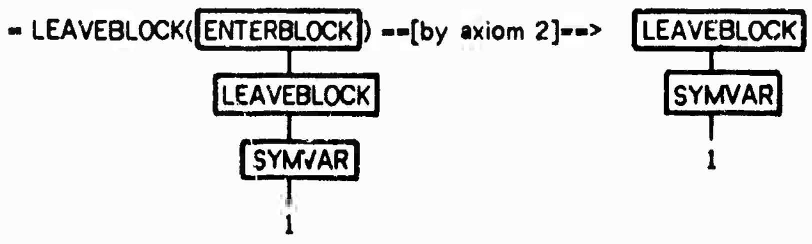


as well as trees built from INIT, ENTERBLOCK, and ADDID nodes.

In general, for a given data type T and operation $F(D_1,...,D_n) \rightarrow T$, we define the *T(F)-expression data type* to be the data type obtained by adding $F(D_1,...,D_n) \rightarrow T$ to the syntactic specification of T. The *T(F)-expression reduction system* is defined to be the maximal direct implementation of the T(F)-expression data type.

The example discussed above is the Symboltable(SYMVAR) Expression data type. The Symboltable(SYMVAR) Expression Reduction System provides not only an implementation of the Symboltable data type, but is also capable of making reductions such as

LEAVEBLOCK(ENTERBLOCK(LEAVEBLOCK(SYMVAR(1))))



= LEAVEBLOCK(SYMVAR(1))

Consequently, this reduction system can be used to carry out steps in proofs about applications of the Symboltable operations: for example, proofs of properties of other parts of a compiler.

5.2 *COMPILATION OF REDUCTION SYSTEMS*

We now turn to a more detailed discussion of how reduction systems (and therefore direct implementations) can be realized by a pattern-match-compilation process (which we shall call PMC). We shall confine ourselves in this report to an informal description in terms of the Symboltable example. From the syntactic specification S of the Symboltable data type, PMC first produces a set of *node constructor operations* INITNODE, ENTERBLOCKNODE, ..., RETRIEVENODE, and *projection* (or *selector*) *operations* ENTERBLOCKA, ADDIDA, ADDIDB, ADDIDC, etc. The constructor operations have the same syntactic specification as the corresponding operations in S, while projection operations are from the constructor range type to one of the argument types of the constructor, e.g.,

ADDIDNODE(Symboltable,Identifier,Attributelist) → Symboltable,
ADDIDA(Symboltable) → Symboltable,
ADDIDB(Symboltable) → Identifier,
ADDIDC(Symboltable) → Attributelist.

The projection operations satisfy the semantic axioms implied by their name , e.g.,

ADDIDA(ADDIDNODE(symtab,id,attrlist)) = symtab,
ADDIDB(ADDIDNODE(symtab,id,attrlist)) = id,
ADDIDC(ADDIDNODE(symtab,id,attrlist)) = attrlist

PMC also constructs a *node discriminator operation*,

SYMTAG(Symboltable) → Integer

such that

SYMTAG(INITNODE) = 0
SYMTAG(ENTERBLOCKNODE(symtab)) = 1
SYMTAG(ADDIDNODE(symtab,id,attrlist)) = 2
etc.

The actual implementation of these constructor, projection, and discriminator operations could of course be any of a variety of implementations which satisfy these axioms, e.g., with Lisp operations:

```
ADDIDNODE(symtab,id,attrlist) = LIST(2,symtab,Id,attrlist),
ADDIDA(symtab) = CADR(symtab),
ADDIDB(symtab) = CADDR(symtab),
ADDIDC(symtab) = CADDDR(symtab),
SYMTAG(symtab) = CAR(symtab).
```

Or, each of these operations could be implemented by machine language code for allocating memory blocks of one or words and accessing fields within these blocks, as discusssed in [Hoare75]. This is the approach taken in Pascal compilers in compiling code for variant record structures, for example. This approach could yield significantly greater efficiency than Lisp operations, but its success depends mainly on having a reasonably efficient garbage collector for variable-size memory blocks.

PMC then proceeds to translate the semantic axioms into programs operating on the nodes. The translation is done incrementally (axiom by axiom). Each operation is first given an initial program definition, just in terms of its corresponding node constructor:

```
INIT = INITNODE
ENTERBLOCK(symtab) = ENTERBLOCKNODE(symtab)
etc.
```

Then, as each axiom is processed, it is used to modify the existing program of one of the operations. The first axiom, for example, LEAVEBLOCK(INIT) = INIT, is used to modify the program for LEAVEBLOCK to be

```
LEAVEBLOCK(symtab) =
     IF SYMTAG(symtab) = 0
          THEN INIT
          ELSE LEAVEBLOCKNODE(symtab)
```

From the next axiom, LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab, PMC produces

```
LEAVEBLOCK(symtab) =
     IF SYMTAG(symtab) = 1
     THEN ENTERBLOCKA(symtab)
          ELSE IF SYMTAG(symtab) = 0
               THEN INIT
               ELSE LEAVEBLOCKNODE(symtab).
```

The third axiom, LEAVEBLOCK(ADDID(symtab,id,attrlist)) = LEAVEBLOCK(symtab), produces

```
LEAVEBLOCK(symtab) =
     IF SYMTAG(symtab) = 2
          THEN LEAVEBLOCK(ADDIDA(symtab))
          ELSE IF SYMTAG(symtab) = 1
               THEN LEAVEBLOCKA(symtab)
               ELSE IF SYMTAG(symtab) = 0
```

```
THEN INIT
ELSE LEAVEBLOCKNODE(symtab).
```

Note the recursive call in this program. The next three axioms lead to a similar program for ISINBLOCK, and the final three determine the RETRIEVE program. (Although the axioms were grouped according to the main operator of the left-hand side, this is not necessary). In general, given an axiom LHS=RHS, PMC modifies the program for F, where F is the main operator of LHS, from its existing program body B to a new body of the form

IF P THEN RHS' ELSE B,

where P is a test for the argument pattern of LHS, and RHS' is obtained from RHS by substituting node projection operations for parameters which occur in patterns in LHS. If no pattern occurs in the arguments of LHS (as in the case of the Stack axiom REPLACE(stk,elm) = PUSH(POP(stk),elm)), the new program body is simply RHS.

5.3 *A CONDITIONAL EVALUATOR (CEVAL)*

CEVAL is an evaluator/simplifier of logical, relational, and arithmetic expressions. It has built-in knowledge of propositional calculus, equality relations, order relations, and arithmetic operations on expressions representing integers and rational numbers. Simplification of arithmetic expressions and pattern matching are handled by calls to the standard evaluator of the Reduce system [Hearn71]. Conditional expressions (IF-THEN-ELSE) are used internally to represent all propositional calculus operators ($\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$). CEVAL implements only a small set of transformations on conditional expressions, similar to those discussed by [McCarthy63] and implemented previously in the Boyer-Moore Theorem Prover [Boyer75]. However, it is still "complete" with respect to propositional calculus in that any valid formula in the propositional calculus (i.e., provable by truth table) will be reduced to TRUE by the transformations. Output is available either in terms of the IF-THEN-ELSE operator or re-expressed in terms of $\neg$, $\wedge$, and $\supset$.

5.3.1 *Propositional calculus rules*

In part, the implementation of CEVAL can be regarded as a Boolean Expression Reduction System obtainable from the Boolean data type specification of Figure 4.1. This reduction system has been extended by including the basic Boolean theorems 1-4 as reduction rules. Additional extensions will be discussed in Section 5.3.2.

We say that a propositional calculus expression is in *simplified IF-THEN-ELSE form* if none of the Boolean axioms or rules 1-4 is applicable to it. The following theorem implies an important completeness property of the set of Boolean axioms and rules 1-4, in that it implies that they are adequate to simplify any valid propositional calculus expression (i.e., one which is true for all possible substitutions of TRUE/FALSE values for its propositional symbols) to TRUE.

Theorem. The only valid propositional calculus expression in simplified IF-THEN-ELSE form is the constant TRUE.

Proof. Assume the theorem is false and let $A(p_1,\ldots,p_n)$ be the smallest expression for which it is false. Then A is a valid expression and can be written

$$\text{IF } B(p_1,\ldots,p_n) \text{ THEN } C(p_1,\ldots,p_n) \text{ ELSE } D(p_1,\ldots,p_n).$$

B cannot itself be TRUE, FALSE, or an IF-THEN-ELSE expression, since if it were then one of the IF-THEN-ELSE axioms or the IF-distribution rule would apply. Therefore, it is some proposition symbol, say p_1. Then C and D must both be independent of p_1, since otherwise one of the two logical substitution rules would apply. Thus A can be written

$$\text{IF } p_1 \text{ THEN } E(p_2,\ldots,p_n) \text{ ELSE } F(p_2,\ldots,p_n).$$

If there existed some assignment of TRUE/FALSE values $p_2',\ldots,p_n'$ to $p_2,\ldots,p_n$ such that $E(p_2',\ldots,p_n')$ evaluated to FALSE, then $\text{TRUE},p_2',\ldots,p_n'$ would be an assignment to $p_1,\ldots,p_n$ for which A would be FALSE, contradicting the validity of A. Therefore, E is valid, and similarly, F is valid. They cannot both be the constant TRUE, since if so the repeated result rule would apply to A. Thus at least one of E or F must be of the IF-THEN-ELSE form (simplified form, since they are part of A). This is a contradiction, since they are smaller than A, which was assumed to be the smallest such expression. *Q.E.D.*

Note that the truth of the theorem actually does not depend on the use of the Redundant IF rule. [McCarthy63] showed that a slightly larger set of rules actually provides a canonical form for propositional calculus expressions, but it does not appear as useful as the simplified IF-THEN-ELSE form, since nonvalid expressions are usually expanded and rearranged in undesirable ways by the canonical form rules.

5.3.2 *Case analysis, equality, and ordering properties*

In addition to the basic knowledge of propositional calculus, CEVAL also includes the case analysis rule of Section 4.1, and equality and ordering rules. Equality substitution is performed by the rules

(IF a = b THEN q[a] ELSE r) = (IF a = b THEN q[b *for* a] ELSE r),
(IF a = b THEN q ELSE r[b = a]) = (IF a = b THEN q ELSE r[FALSE *for* b = a]).

(See Section 4.1 for an explanation of the brackets in these rules.) Note that occurrences of the predicate (a = b) in the THEN or ELSE parts are eliminated by the logical substitution rules (Section 4.1).

Aside from these substitution properties, CEVAL also uses the reflexive property of the equality operator, (a = a) = TRUE. This combined with the substitution properties also gives the symmetric property, $(a = b) \supset (b = a)$, and the transitivity property, $(a = b) \wedge (b = c) \supset (a = c)$. The not-equal operator is defined by the axiom, $(a \neq b) = \neg(a = b)$ (which internally is IF a = b THEN FALSE ELSE TRUE).

The following transformations give CEVAL a (very limited) knowledge of ordering properties:

$(a \geq b) = (b \leq a)$,
$(a < b) = \neg(b \leq a)$,
$(a > b) = \neg(a \leq b)$,
(IF $a \leq b$ THEN q[$b \leq a$] ELSE r) = (IF $a \leq b$ THEN q[$a = b$ *for* $b \leq a$] ELSE r),
(IF $a \leq b$ THEN q ELSE r[$b \leq a$]) = (IF $a \leq b$ THEN q ELSE r[TRUE *for* $b \leq a$]),

$(\text{IF } a \leq b \text{ THEN } q \text{ ELSE } r[a = b]) = (\text{IF } a \leq b \text{ THEN } q \text{ ELSE } r[\text{FALSE } \textit{for } a = b]),$
$(\text{IF } a \leq b \text{ THEN } q \text{ ELSE } r[b = a]) = (\text{IF } a \leq b \text{ THEN } q \text{ ELSE } r[\text{FALSE } \textit{for } b = a]).$

5.3.3 *Translation back to usual logical operators*

Although the IF-THEN-ELSE form of logical expressions has many advantages for simplification, use of equalities, etc., it can be difficult to read if used as the final output of the simplification process. Thus CEVAL provides for a translation back to the more familiar logical operators $\wedge$, $\neg$, and $\supset$. Using the fact that the IF-THEN-ELSE expression to be translated is in simplified form, we need only a small set of transformations to put it in terms of the desired operators:

$(\text{IF } p \text{ THEN } q \text{ ELSE TRUE}) = (p \supset q),$
$(\text{IF } p \text{ THEN } q \text{ ELSE FALSE}) = (p \wedge q),$
$(\text{IF } p \text{ THEN TRUE ELSE } q) = (\neg p \supset q),$
$(\text{IF } p \text{ THEN FALSE ELSE } q) = (\neg p \wedge q),$
$(p \supset \text{FALSE}) = \neg p,$
$(p \supset (q \supset r)) = ((p \wedge q) \supset r),$
$(p \wedge \text{TRUE}) = p,$
$\neg(a \leq b) = (b < a),$
$\neg(a = b) = (a \neq b).$

If neither q nor r in (IF p THEN q ELSE r) is equal to TRUE or FALSE, the IF-THEN-ELSE operator is retained.

5.4 *A DATA TYPE VERIFICATION SYSTEM (DTVS)*

The complete implementation of the symbol table operations by a stack of hash tables has in fact been verified using the verification system at ISI. Because the user interface for the proof process is still under development, we will not give a detailed description of the commands of the current facilities (called the Data Type Verification System), but merely indicate the nature of the commands and overall proof process with an example, the verification of the top level implementation by a Stack of Mappings. The first step is to direct the system to adopt the programs of data type Symboltable, and the axioms of data types Stack and Mapping. These programs and axioms would all be in the form of rewrite rules which the user had just entered or had read in from files. The command for "adopting" a set of rules is separated from the act of reading them in so that several sets of rules for an operator can coexist within the system. Assuming the Symboltable axioms have also been input to the system, the user then directs the system to generate the verification conditions for the data type. These would consist of the Symboltable axioms and the equality axioms for the Symboltable equality operator (see Section 4.4), all interpreted in terms of the representation.

The user can then attempt to prove each of the verification conditions using CEVAL or the standard simplifier/theorem prover of the system. In these proofs the rewrite rules from the Symboltable programs and Stack and Mapping axioms are used automatically, without further direction from the user. In some cases, as noted in Section 4, completion of a proof will require one or more assumptions to be made about the representation or the Stack or Mapping data types. If this is the case, the system will stop with a reduced form of the original verification condition. Examination of this output will often lead the user to the necessary assumptions, which are initially input by the user and used as needed without justification. To complete the verification of the implementation, it is necessary to prove these assumptions, or a stronger set of assumptions, as invariants (of the Symboltable data type implementation or of the Stack or Mapping data types). The verification conditions sufficient to establish these invariants are constructed using the syntactic specifications of the data types, in accordance with the principle of data type induction, as described in Section 4.3.

6. *SOME EXTENSIONS AND CONCLUSIONS*

This section has two purposes: to briefly discuss a few miscellaneous issues not treated in earlier sections and to try and put the rather large amount of material presented in these sections into some perspective. We begin with two subsections, each dealing with a related set of problems.

6.1 *PROCEDURES*

The most common example in the literature on algebraic specification is undoubtedly type Stack:

```
type Stack

syntax

    NEWSTACK → Stack,
    PUSH(Stack, Integer) → Stack,
    POP(Stack) → Stack,
    TOP(Stack) → Integer.

semantics

  declare s:Stack, i:Integer;
    POP(NEWSTACK) = NEWSTACK,
    POP(PUSH(s,i)) = s,
    TOP(NEWSTACK) = UNDEFINED,
    TOP(PUSH(s,i)) = i.
```

From a pedagogical point of view this example has much to recommend it: it is consistent, complete (by almost any definition), concise, and easy to understand. It has one major drawback--the operations axiomatized are not those most programmers associate with stacks. The operators defined above are purely functional, that is to say they are mappings from a cross-product of values to a value. To preserve the value generated by applying one of the operators to values, one must use an assignment operator defined outside type Stack. A typical program segment might look like

```
declare s:Stack, i:Integer;
s ← NEWSTACK;
s ← PUSH(s,3);
i ← TOP(s);
s ← POP(s);
```

This complete dependence on pure functions and assignment is foreign to the way most people program. Almost all modern programming languages allow their users to define procedures that directly alter at least some of their parameters. Thus we find program segments of the form

```
declare s:Stack, i:Integer
s ← NEWSTACK
PUSHON(s,3);
i ← POPTOP(s);
```

where PUSHON alters its first parameter and POPTOP not only returns a value but also has an effect on its parameter. Procedures that alter their parameters are useful for a number of reasons, and therefore any good specification technique should be capable of dealing with them.

We begin by changing the syntactic specifications so that they distinguish between variable and constant parameters. The syntactic specification for type Stack will then include

```
PUSHON(var Stack, Integer) →
POPTOP(var Stack) → Integer
```

The first line tells us that PUSHON is a pure procedure (it does not have a value) that alters its first argument but not its second (const is the default). The second line tells us POPTOP is a function procedure that modifies its parameter and, when applied to a stack, has a value of its own. These lines do not, of course, tell us anything about how the parameters are altered or what values POPTOP(s) is to have. Our approach to providing this information is closely related to work on the axiomatic specification of the meaning of procedures in programming languages.

If one assumes no implicit parameters (i.e., global variables), Hoare and Wirth's discussion [Hoare73] of the meaning of the procedure declaration *procedure* p(L):S may be paraphrased as follows:

Let x be the list of explicit parameters declared in L; let $x_1,...,x_m$ be the parameters declared in L as variable parameters. Given the assertion Q{S}R we may define the existence of functions F_i satisfying the implication: Q ⊃ R with each x_i replaced by F_i. The functions F_i may be regarded as those which map the initial values of x on entry to the procedure onto the final values of $x_1,...,x_m$ on completion of the execution of S. What the above says is that the meaning of the procedure p may be expressed in terms of the simultaneous assignment to its *var* parameters of the values obtained by applying the functions F_i to the parameters passed to p. The programs computing these functions can, it says, be deduced from S, the body of p.

Taking our cue from this rule, we have extended our specification technique by allowing operators that alter their parameters to be defined in terms of functions that do not. We thus have axioms such as

```
PUSHON(s,i) = s ← PUSH(s,i),
POPTOP(s) = s ← POP(s); TOP(s).
```

where ← is a (simultaneous) assignment operator and the expression to the right of the semicolon (TOP(s) in the second axiom) is the value returned by the procedure. It is important to note that the expression to the right of the semicolon is evaluated at the same time as the simultaneous assignments. Thus the assignments have no effect on the value returned. If the specification also includes sufficient axioms to define PUSH, POP, and TOP the above axioms are sufficient to fully define PUSHON and POPTOP. A sufficiently complete axiomatization of a type Stack with operations PUSHON and POPTOP follows:

type Stack

syntax

```
  NEWSTACK → Stack,
 *PUSH(Stack,Integer) → Stack,
 *POP(Stack) → Stack,
 *TOP(Stack) → Integer,
  PUSHON(var Stack, Integer),
  POPTOP(var Stack) → Integer.
```

semantics

```
 declare s:Stack, i:Integer
  POP(NEWSTACK) = NEWSTACK,
  POP(PUSH(s,i)) = s,
  TOP(NEWSTACK) = UNDEFINED,
  TOP(PUSH(s,i,)) = i,
  PUSHON(s,i) = s ← PUSH(s,i),
  POPTOP(s) = s ← POP(s); TOP(s).
```

The asterisk preceding some of the functions in the syntactic specification indicates that those operators are "hidden" functions which facilitate the definition of the other operators. They are not available to users of the abstract type Stack, nor need they be implemented.

Our approach to proving the correctness of implementations of procedures such as PUSHON and POPTOP derives from the proof rule for procedure declarations. The first step is to derive from the body of the procedure programs implementing the functions F_i. From an implementation for PUSHON, for example, we derive one F_i corresponding to PUSH. Once this has been done we convert these programs to recursively defined functions using techniques described in [McCarthy63] and [Manna74]. We now need only show that all of the derived F_i are correct implementations of the axiomatized operations they correspond to. We would, for example, have to show that the derived implementations of PUSH, POP, and TOP combine with the programmer supplied implementation of NEWSTACK to form a model for the first four axioms. How to do this has been discussed at length in Section 4.

6.2 *ERRORS*

It is a rare data type that has nothing but "total" operators defined for it. Both the traditionally built-in types (e.g., Integer) and most user-defined types (e.g., Symboltable) have operators that are not well defined over all of the values of the type. Programmers are all too familiar with such unwelcome messages as "attempt to divide by zero" and "symbol table overflow." Throughout this paper we have consistently dealt with such "partial" functions by explicitly supplying distinguished values, e.g., UNDEFINED for those instances in which one might normally think of the function as having no value. We have done this for two reasons. First, it allows us to treat all operators as total, an assumption that simplifies the theory underlying the applications discussed in this paper. Second, by forcing the authors of a specification to explicitly deal with all syntactically legal uses of the operators we cut down the incidence of errors of omission.

Our use of the UNDEFINED value in this paper has been relatively light. We used examples in which the mechanism for specifying error conditions would not detract from the points we were using the examples to illustrate. In practice we often wish to define types in which errors play a more prominent role. Among the most common examples are data types of limited size (as opposed to those we have used up to now, which grow without bound). Our experience with types such as these has led us to revise our specifications to include a special mechanism for handling error conditions. The following simple example illustrates some of the inadequacies of the approach to errors taken in some of the earlier work, e.g., [Guttag75], on algebraic specifications.

```
type Bstack

syntax

          NEWSTACK(Integer) → Bstack,
          PUSH(Bstack,Integer) → Bstack ∪ {ERROR},
          POP(Bstack) → Bstack ∪ {ERROR},
          TOP(Bstack) → Integer ∪ {UNDEFINED} ∪ {ERROR},
          SIZE(Bstack) → Integer,
          LIMIT(Bstack) → Integer.

semantics

     declare i:Integer,s:Bstack;
          POP(NEWSTACK(i))=NEWSTACK(i),
          POP(PUSH(s,i))=IF SIZE(s) < LIMIT(s)
                                  THEN s
                                  ELSE ERROR,
          TOP(NEWSTACK(i))=UNDEFINED,
          TOP(PUSH(s,i))=IF SIZE(s) < LIMIT(s)
                                  THEN i
                                  ELSE ERROR,
          LIMIT(NEWSTACK(i))=i,
          LIMIT(PUSH(s,i))=LIMIT(s),
          SIZE(NEWSTACK(i))=0,
          SIZE(PUSH(s,i))=SIZE(s)+1.
```

Notice that the ranges of some of the operations have been augmented with the singleton sets {UNDEFINED} or {ERROR}. One could avoid doing this by assuming that these distinguished values are implicitly included in all types, but then one can no longer always assume that the axioms are universally quantified over the types. A more fundamental problem associated with this specification is that it does not accurately parallel most people's concept of a bounded stack. Stack overflow occurs not when we try to push one too many items onto the stack, but rather when we attempt to perform a POP, TOP, or SIZE operation upon a stack onto which too many items have been pushed.

These problems cannot be cured by the simple expedient of adding the axiom beginning

PUSH(s,i)=IF SIZE(s) ≥ LIMIT(s) THEN ERROR

First, it is not clear that there is any meaningful value we can use in the ELSE clause. We might consider merely repeating the left-hand side in the ELSE clause; this would serve to indicate that PUSH can generate an error. It would not, however, allow us to eliminate the test in axioms 2 and 4, for if we did that it would leave us with an inconsistent axiom set. The introduction of such "circular" axioms would also serve to complicate the processes of automatic verification and the generation of direct implementations.

One solution to this problem is to introduce intermediate functions. Rather than let the user directly invoke functions that may not always be well-defined, we mark these functions as hidden and interpose some sort of access operation between them and the user. The purpose of this access operation is to ensure that any hidden function is invoked only with arguments for which it will be well-defined. The hidden function may then be assumed to be "total" in the sense that it is well-defined for all values to which it can be applied. The following specification of type Bstack, for example, may be construed as sufficiently complete despite the fact that TOP and PUSH are not everywhere defined.

type Bstack

syntax

```
        NEWSTACK(Integer) → Bstack,
       *PUSH(Bstack,Integer) → Bstack,
        POP(Bstack) → Bstack,
       *TOP(Bstack) → Integer,
        SIZE(Bstack) → Integer,
        LIMIT(Bstack) → Integer,
        PUSHON(Bstack,Integer) → Bstack ∪ {ERROR},
        TOPOF(Bstack) → Integer ∪ {UNDEFINED}.
```

semantics

```
    declare i:Integer,s:Bstack;
        POP(NEWSTACK(i))=NEWSTACK(i),
        POP(PUSH(s,i))=s,
        TOP(PUSH(s,i))=i,
        LIMIT(NEWSTACK(i))=i,
        LIMIT(PUSH(s,i))=LIMIT(s),
        SIZE(NEWSTACK(i))=0,
        SIZE(PUSH(s,i))=SIZE(s)+1,
        PUSHON(s,i)=IF SIZE(s) < LIMIT(s)
                         THEN PUSH(s,i)
                         ELSE ERROR,
        TOPOF(s)=IF SIZE(s)=0
                         THEN UNDEFINED
                         ELSE TOP(s).
```

This approach presents no technical problems. Nevertheless, we do not feel that it is entirely adequate in all situations. While the extra level of nesting in axioms 8 and 9 and the introduction of the intermediary operations PUSHON and TOPOF does not seem to have an overly severe effect on the clarity of this specification, this is not always the case. If a large number of operations can cause errors, the specification can become large and unwieldy. In order to understand how the operations behave under normal circumstances one must first wade through a specification of what is to happen in the exceptional cases involving errors; this has led us to allow for factoring out the exceptional conditions from the main body of the procedure. One way to do this is to associate with each operator an entry condition defining those values to which it is permissible to apply that operator. We have adopted a related approach in which every specification has, in addition to a syntactic and a semantic specification, a *restriction specification* that tells us when the value of an operation will not be well-defined. This leads to the following specification of type Bstack:

type Bstack

syntax

```
NEWSTACK(Integer) → Bstack,
PUSH(Bstack,Integer) → Bstack ∪ {ERROR},
POP(Bstack) → Bstack,
TOP(Bstack) → Integer ∪ {UNDEFINED},
SIZE(Bstack) → Integer,
LIMIT(Bstack) → Integer.
```

semantics

declare i:Integer,s:Bstack;

```
POP(NEWSTACK(i)) = NEWSTACK(i),
POP(PUSH(s,i)) = s,
TOP(PUSH(s,i) = i,
LIMIT(NEWSTACK(i)) = i,
LIMIT(PUSH(s,i)) = LIMIT(s),
SIZE(NEWSTACK(i)) = 0,
SIZE(PUSH(s,i)) = 1+SIZE(s),
```

restrictions

```
SIZE(s) ≥ LIMIT(s) ⊃ PUSH(s,i) = ERROR,
s = NEWSTACK(i) ⊃ TOP(s) = UNDEFINED.
```

Note that the main body of this specification is somewhat simpler than our earlier specification of this type. Note too that the restriction specification is quite short, which reflects the fact that most operations are permissible; a "permissible specification" would be quite a bit longer.

The implications in the restriction specification have been augmented by a value that is to be returned if the predicate is true; this allows the author of a specification to distinguish among the various types of errors that may occur. It may be possible to prove, at compile time, that one or more of the predicates in the restriction specification is always false at the time the operation in question is invoked. If this is not the case, one must program a runtime check. One way to do this is to add a NEWSTACK operation to the type, e.g., PUSH-OK, that corresponds to the restriction condition (or its inverse). The syntactic specification of type Bstack might then be augmented by

PUSH-OK(Bstack) → Boolean

and the semantic specification by

PUSH-OK(s) = (SIZE(s) < LIMIT(S))

6.3 *CONCLUSIONS*

Elsewhere it has been argued that abstract data types can be effectively employed as a "thought tool" in the structured development of programs ([Guttag 76b], [Standish 73], [Wulf 76], [Liskov 75]). In this report we have attempted to show that the use of algebraic axioms as a means for describing data abstractions is also valuable, when properly used, for both formal and informal program validation.

In Sections 2 and 3 we discussed the axiom language and some techniques for constructing algebraic axiomatizations. A complete implementation of a moderately complex symbol table was developed to show how these specifications might be used in practice. In Section 4 we demonstrated how properly written axioms can be used in formal program verification. Abstract data types provide a mechanism for factoring proofs into manageable sections. At one level of abstraction the axiomatic specification provides us with theorems that may be applied in the verification of programs that use abstract data types. Writing the axioms in a certain style allows them to be used as reduction rules so that the proofs become largely symbol manipulation exercises. We have illustrated how such axioms may be used to verify the correctness of implementations of higher-level abstract data types in terms of lower-level ones.

In Section 5 we have formalized the symbol manipulation processes needed to prove correctness and to provide direct implementations. This system is capable of carrying out large parts of the proofs without the need of a theorem prover; therefore, the cost of execution plus the software sophistication normally required for such proofs is substantially reduced. Furthermore, this system is able to exploit the duality between axioms and programs as described in Section 3. A direct implementation of a data type is achieved by using the axioms. This allows for the interpretive execution of programs which make use of an algebraically axiomatized data type. The coupling of early testing with proofs of correctness within the same automated framework is a valuable tool in the software production process.

Throughout this report we have expressed great optimism about the role of abstract data types and their algebraic specifications in program development and validation. It is important to notice that the techniques developed in this paper are essentially programming language independent. While the availability of languages with the compile time type facilities of SIMULA 67 [Dahl 68], CLU [Liskov 74], or Alphard [Wulf 76] will make these techniques easier to apply, they are by no means essential. If one exercises enough self discipline to ensure the validity of data type induction, the techniques described should prove useful in the development of programs in any language. The ultimate test of any research designed to enhance the process of software development must lie in the application of the results of that research to a large software project. We feel that the work described in this report lays the groundwork necessary for such a test.

ACKNOWLEDGMENTS

We would like to thank Dick Jenks, Ralph London, Nancy Lynch, Mark Moriconi, Jernej Polajnar, Dave Wile, and Marty Yonke for their careful reading of earlier drafts of this paper, and Betty Randall for her help in typing tho manuscript.

REFERENCES

[Boyer75] Boyer, R. S., and J S. Moore, "Proving theorems about LISP functions," *J. ACM*, 22, 1, January 1975, 129-144.

[Dahl68] Dahl, O.-J. , *The SIMULA 67 common base language*, Norwegian Computing Center, Oslo, 1968.

[Goguen75] Goguen, J. A., J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Abstract data-types as initial algebras and correctness of data representations," *Proceedings*, Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975.

[Good75] Good, D. I., R. L. London, and W. W. Bledsoe. "An interactive program verification system," *IEEE Transactions on Software Engineering*, SE-1, 1, March 1975, 56-67.

[Griesmer71] Griesmer, J. H., and R. D. Jenks, SCRATCHPAD/1--An interactive facility for symbolic mathematics," *Proceedings, Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 42-58.

[Guttag75] Guttag, J. V., "The specification and application to programming of abstract data types," Ph. D. Thesis, University of Toronto, Department of Computer Science, 1975, available as Computer System Research Report CSRG-59.

[Guttag76a] Guttag, J. V., "Abstract data types and the development of data structures," Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, March 1976, pp. 37-46.

[Guttag76b] Guttag, J. V., E. Horowitz, and D. Musser, "The design of data type specifications," *USC Information Sciences Institute Research Report*, 1976 (to appear in Proc. Second International Conference on Software Engineering, San Francisco, October 1976).

[Hearn71] Hearn, A. C., "Reduce 2: a system and language for algebraic manipulation," *Proceedings Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 128-133.

[Hoare72] Hoare, C.A.R., "Proof of correctness of data representations, "*Acta Informatica*, 4, 1972, pp. 271-281.

[Hoare73] Hoare, C.A.R., and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Informatica*, 2, 1973, pp. 335-355.

[Hoare75] Hoare, C.A.R., "Recursive data structures," *International J. of Computer and Information Sciences*, 4,2, June 1975, 105-132.

[Horowitz76] Horowitz, E., and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, June 1976.

[Jenks74] Jenks, R. D., "The SCRATCHPAD language," *Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices*, 9, 4, April 1974, 101-111.

[Liskov74] Liskov, B. H., and S. N. Zilles, "Programming with abstract data types," *Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices*, 9, 4, April 1974, 50-59.

[Liskov75] Liskov, B. H., and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 7-18.

[Martin71] Martin, W. A., and R. J. Fateman, "The MACSYMA system," *Proceedings Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 59-75.

[Manna74] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[McCarthy63] McCarthy, J., "Basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirchberg (eds.), North-Holland Publishing Company, 1963, pp. 33-70.

[Parnas72] Parnas, D. L., "Information distribution aspects of design methodology," Proc. IFIP Congress 71, Vol. 1 (1972), pp. 339-344.

[Rosen73] Rosen, B., "Tree manipulating systems and Church-Rosser theorems," *J. ACM*, 20,1, January 1973, 160-187.

[Spitzen75] Spitzen, J., and B. Wegbreit, "The verification and synthesis of data structures," *Acta Informatica* 4, 127-144 (1975).

[Standish73] Standish, T. A., "Data structures: an axiomatic approach," BBN Report No. 2639, Bolt Beranek and Newmann, Cambridge, Mass., 1973.

[Wegbreit76] Wegbreit, B., and J. Spitzen, "Proving properties of complex data structures," *J. ACM*, 23, 2, April 1976, 389-396.

[Wulf76] Wulf, W. A., R. L. London, and M. Shaw, "Abstraction and verification in Alphard: introduction to language and methodology," *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Zilles75] Zilles, S. N., "Abstract specifications for data types," IBM Research Laboratory, San Jose, California, 1975.

DISTRIBUTION LIST

Director
Defense Advanced Research Projects Agency
Attn: Program Management
1400 Wilson Blvd., Arlington, VA 22209

Defense Documentation Center
Cameron Station
Alexandria, VA 22314

Steven Swart, ACO
Office of Naval Research
1030 Green Street
Pasadena, CA 91106